

Want processes to co-exist

<table>
<thead>
<tr>
<th>Process</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>0x9000</td>
</tr>
<tr>
<td>gcc</td>
<td>0x7000</td>
</tr>
<tr>
<td>bochs/pintos</td>
<td>0x4000</td>
</tr>
<tr>
<td>emacs</td>
<td>0x3000</td>
</tr>
<tr>
<td></td>
<td>0x0000</td>
</tr>
</tbody>
</table>

- Consider multiprogramming on physical memory
  - What happens if emacs needs to expand?
  - If emacs needs more memory than is on the machine??
  - If emacs has an error and writes to address 0x7100?
  - When does gcc have to know it will run at 0x4000?
  - What if emacs isn’t using its memory?
Issues in sharing physical memory

- **Protection**
  - A bug in one process can corrupt memory in another
  - Must somehow prevent process A from trashing B’s memory
  - Also prevent A from even observing B’s memory (ssh-agent)

- **Transparency**
  - A process shouldn’t require particular physical memory bits
  - Yes processes often require large amounts of contiguous memory (for stack, large data structures, etc.)

- **Resource exhaustion**
  - Programmers typically assume machine has “enough” memory
  - Sum of sizes of all processes often greater than physical memory
Virtual memory goals

- Give each program its own “virtual” address space
  - At run time, Memory-Management Unit relocates each load, store to actual memory… App doesn’t see physical memory

- Also enforce protection
  - Prevent one app from messing with another’s memory

- And allow programs to see more memory than exists
  - Somehow relocate some memory accesses to disk
Virtual memory advantages

- Can re-locate program while running
  - Run partially in memory, partially on disk
- Most of a process’s memory may be idle (80/20 rule).
  
  - Write idle parts to disk until needed
  - Let other processes use memory of idle part
  - Like CPU virtualization: when process not using CPU, switch
    (Not using a memory region? switch it to another process)
- Challenge: VM = extra layer, could be slow
Idea 1: load-time linking

- **Linker** patches addresses of symbols like `printf`
- **Idea**: link when process executed, not at compile time
  - Determine where process will reside in memory
  - Adjust all references within program (using addition)

- **Problems?**
Idea 1: load-time linking

- **Linker** patches addresses of symbols like `printf`
- Idea: link when process executed, not at compile time
  - Determine where process will reside in memory
  - Adjust all references within program (using addition)
- **Problems?**
  - How to enforce protection
  - How to move once already in memory (Consider: data pointers)
  - What if no contiguous free region fits program?
Idea 2: base + bound register

- Two special privileged registers: base and bound

- On each load/store:
  - Physical address = virtual address + base
  - Check $0 \leq$ virtual address $< \text{bound}$, else trap to kernel

- How to move process in memory?

- What happens on context switch?
Idea 2: base + bound register

- Two special privileged registers: base and bound
- On each load/store:
  - Physical address = virtual address + base
  - Check $0 \leq$ virtual address $<$ bound, else trap to kernel
- How to move process in memory?
  - Change base register
- What happens on context switch?
Idea 2: base + bound register

- Two special privileged registers: base and bound
- On each load/store:
  - Physical address = virtual address + base
  - Check $0 \leq$ virtual address $< \text{bound}$, else trap to kernel
- How to move process in memory?
  - Change base register
- What happens on context switch?
  - OS must re-load base and bound register
Definitions

- Programs load/store to **virtual** (or **logical**) addresses
- Actual memory uses **physical** (or **real**) addresses
- VM Hardware is Memory Management Unit (**MMU**)
Address space

Virtual Address View

Physical Address View

MMU

OS
Base+bound trade-offs

- **Advantages**
  - Cheap in terms of hardware: only two registers
  - Cheap in terms of cycles: do add and compare in parallel
  - Examples: Cray-1 used this scheme

- **Disadvantages**
Base+bound trade-offs

• **Advantages**
  - Cheap in terms of hardware: only two registers
  - Cheap in terms of cycles: do add and compare in parallel
  - Examples: Cray-1 used this scheme

• **Disadvantages**
  - Growing a process is expensive or impossible
  - No way to share code or data (E.g., two copies of bochs)

• **One solution: Multiple segments**
  - E.g., separate code, stack, data segments
  - Possibly multiple data segments
Let processes have many base/bound regs
- Address space built from many segments
- Can share/protect memory at segment granularity

Must specify segment as part of virtual address
• Each process has a segment table

• Each VA indicates a segment and offset:
  - Top bits of addr select segment, low bits select offset (PDP-10)
  - Or segment selected by instruction or operand (means you need wider “far” pointers to specify segment)
• 2-bit segment number (1st digit), 12 bit offset (last 3)

- Where is 0x0240? 0x1108? 0x265c? 0x3002? 0x1600?
Segmentation trade-offs

• Advantages
  - Multiple segments per process
  - Allows sharing! (how?)
  - Don’t need entire process in memory

• Disadvantages
  - Requires translation hardware, which could limit performance
  - Segments not completely transparent to program (e.g., default segment faster or uses shorter instruction)
  - $n$ byte segment needs $n$ contiguous bytes of physical memory
  - Makes fragmentation a real problem.
• **Fragmentation** $\implies$ Inability to use free memory

• **Over time:**
  - Variable-sized pieces = many small holes (external fragmentation)
  - Fixed-sized pieces = no external holes, but force internal waste (internal fragmentation)
Alternatives to hardware MMU

- **Language-level protection (Java)**
  - Single address space for different modules
  - Language enforces isolation
  - Singularity OS does this [Hunt]

- **Software fault isolation**
  - Instrument compiler output
  - Checks before every store operation prevents modules from trashing each other
  - Google Native Client does this with only about 5% slowdown [Yee]
Paging

- Divide memory up into small \textit{pages}
- Map virtual pages to physical pages
  - Each process has separate mapping
- Allow OS to gain control on certain operations
  - Read-only pages trap to OS on write
  - Invalid pages trap to OS on read or write
  - OS can change mapping and resume application
- Other features sometimes found:
  - Hardware can set "accessed" and "dirty" bits
  - Control page execute permission separately from read/write
  - Control caching or memory consistency of page
Paging trade-offs

- Eliminates external fragmentation
- Simplifies allocation, free, and backing storage (swap)
- Average internal fragmentation of .5 pages per "segment"
• Allocate any physical page to any process
• Can store idle virtual pages on disk
Paging data structures

- Pages are fixed size, e.g., 4K
  - Least significant 12 (log₂ 4K) bits of address are page offset
  - Most significant bits are page number

- Each process has a page table
  - Maps virtual page numbers (VPNs) to physical page numbers (PPNs)
  - Also includes bits for protection, validity, etc.

- On memory access: Translate VPN to PPN, then add offset
Example: Paging on PDP-11

- 64K virtual memory, 8K pages
  - Separate address space for instructions & data
  - I.e., can’t read your own instructions with a load

- Entire page table stored in registers
  - 8 Instruction page translation registers
  - 8 Data page translations

- Swap 16 machine registers on each context switch
x86 Paging

- Paging enabled by bits in a control register (%cr0)
  - Only privileged OS code can manipulate control registers

- Normally 4KB pages

- %cr3: points to 4KB page directory

- Page directory: 1024 PDEs (page directory entries)
  - Each contains physical address of a page table

- Page table: 1024 PTEs (page table entries)
  - Each contains physical address of virtual 4K page
  - Page table covers 4 MB of Virtual mem

- See intel manual for detailed explanation
  - Volume 2 of AMD64 Architecture docs
  - Volume 3A of Intel Pentium Manual
x86 page translation

32 bits aligned onto a 4-KByte boundary

1024 PDE \times 1024 PTE = 2^{20} \text{ Pages}

*32 bits aligned onto a 4-KByte boundary
## x86 page directory entry

### Page–Directory Entry (4-KByte Page Table)

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page–Table Base Address</td>
<td>0–31</td>
</tr>
<tr>
<td>Available</td>
<td>31</td>
</tr>
<tr>
<td>Global</td>
<td>30–28</td>
</tr>
<tr>
<td>Page size</td>
<td>27–20</td>
</tr>
<tr>
<td>Reserved</td>
<td>19–13</td>
</tr>
<tr>
<td>Accessed</td>
<td>12–11</td>
</tr>
<tr>
<td>Cache disabled</td>
<td>10</td>
</tr>
<tr>
<td>Write-through</td>
<td>9</td>
</tr>
<tr>
<td>User/Supervisor</td>
<td>8</td>
</tr>
<tr>
<td>Read/Write</td>
<td>7–5</td>
</tr>
<tr>
<td>Present</td>
<td>4–0</td>
</tr>
</tbody>
</table>

- **Available** for system programmer’s use
- **Global page** (Ignored)
- **Page size** (0 indicates 4 KBytes)
- **Reserved** (set to 0)
- **Accessed**
- **Cache disabled**
- **Write-through**
- **User/Supervisor**
- **Read/Write**
- **Present**
# x86 page table entry

Page-Table Entry (4-KByte Page)

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Avail</td>
<td>G</td>
<td>P</td>
<td>A</td>
<td>T</td>
<td>D</td>
<td>A</td>
<td>P</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

- **Available for system programmer’s use**
- **Global Page**
- **Page Table Attribute Index**
- **Dirty**
- **Accessed**
- **Cache Disabled**
- **Write-Through**
- **User/Supervisor**
- **Read/Write**
- **Present**
x86 hardware segmentation

- **x86 architecture also supports segmentation**
  - Segment register base + pointer val = *linear address*
  - Page translation happens on linear addresses

- **Two levels of protection and translation check**
  - Segmentation model has four privilege levels (CPL 0–3)
  - Paging only two, so 0–2 = kernel, 3 = user

- **Why do you want both paging and segmentation?**

  - Short answer: You don’t – just adds overhead
    - Most OSes use “flat mode” – set base = 0, bounds = 0xffffffff in all segment registers, then forget about it

  - Long answer: Has some fringe/incidental uses
    - VMware runs guest OS in CPL 1 to trap stack faults
    - OpenBSD used CS limit for W ∧ X when no PTE NX bit
x86 hardware segmentation

- x86 architecture also supports segmentation
  - Segment register base + pointer val = \textit{linear address}
  - Page translation happens on linear addresses

- Two levels of protection and translation check
  - Segmentation model has four privilege levels (CPL 0–3)
  - Paging only two, so 0–2 = kernel, 3 = user

- Why do you want both paging and segmentation?

- Short answer: You don’t – just adds overhead
  - Most OSes use “flat mode” – set base = 0, bounds = 0xffffffff in all segment registers, then forget about it
  - x86-64 architecture removes much segmentation support

- Long answer: Has some fringe/incidental uses
  - VMware runs guest OS in CPL 1 to trap stack faults
  - OpenBSD used CS limit for \(W\land X\) when no PTE NX bit
Making paging fast

- x86 PTs require 3 memory references per load/store
  - Look up page table address in page directory
  - Look up PPN in page table
  - Actually access physical page corresponding to virtual address

- For speed, CPU caches recently used translations
  - Called a translation lookaside buffer or TLB
  - Typical: 64-2K entries, 4-way to fully associative, 95% hit rate
  - Each TLB entry maps a VPN → PPN + protection information

- On each memory reference
  - Check TLB, if entry present get physical address fast
  - If not, walk page tables, insert in TLB for next time
    (Must evict some entry)
TLB details

- TLB operates at CPU pipeline speed $\Rightarrow$ small, fast
- Complication: what to do when switch address space?
  - Flush TLB on context switch (e.g., old x86)
  - Tag each entry with associated process's ID (e.g., MIPS)
- In general, OS must manually keep TLB valid
- E.g., x86 `invlpg` instruction
  - Invalidates a page translation in TLB
  - Must execute after changing a possibly used page table entry
  - Otherwise, hardware will miss page table change
- More Complex on a multiprocessor (TLB shootdown)
x86 Paging Extensions

• **PSE: Page size extensions**
  - Setting bit 7 in PDE makes a 4MB translation (no PT)

• **PAE Page address extensions**
  - Newer 64-bit PTE format allows 36 bits of physical address
  - Page tables, directories have only 512 entries
  - Use 4-entry Page-Directory-Pointer Table to regain 2 lost bits
  - PDE bit 7 allows 2MB translation

• **Long mode PAE**
  - In Long mode, pointers are 64-bits
  - Extends PAE to map 48 bits of virtual address (next slide)
  - Why are aren’t all 64 bits of VA usable?
x86 long mode paging

Virtual Address

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>48</td>
<td>47</td>
<td>39 38</td>
<td>30 29</td>
<td>21 20</td>
</tr>
</tbody>
</table>

- 9-bit: Page-Map Level-4 Table
- 9-bit: Page Directory Pointer Table
- 9-bit: Page Directory Table
- 9-bit: Page Table
- 12-bit: 4-Kbyte Physical Page

- 12-bit: Page-Map L4 Base Addr
- 52-bit: Physical Address
- 51-bit: CR3

Virtual Address

Sign Extend: 63-bit
Page-Map Level-4 offset (PML4): 48-bit
Page Directory Pointer Offset: 47-bit
Page Directory Offset: 39-bit 38-bit
Page-Table Offset: 30-bit 29-bit
Physical-Page Offset: 21-bit 20-bit
Where does the OS live?

• In its own address space?
  - Can’t do this on most hardware (e.g., syscall instruction won’t switch address spaces)
  - Also would make it harder to parse syscall arguments passed as pointers

• So in the same address space as process
  - Use protection bits to prohibit user code from writing kernel

• Typically all kernel text, most data at same VA in every address space
  - On x86, must manually set up page tables for this
  - Usually just map kernel in contiguous virtual memory when boot loader puts kernel into contiguous physical memory
  - Some hardware puts physical memory (kernel-only) somewhere in virtual address space
Very different MMU: MIPS

- Hardware has 64-entry TLB
  - References to addresses not in TLB trap to kernel

- Each TLB entry has the following fields:
  Virtual page, Pid, Page frame, NC, D, V, Global

- Kernel itself unpaged
  - All of physical memory contiguously mapped in high VM
  - Kernel uses these pseudo-physical addresses

- User TLB fault handler very efficient
  - Two hardware registers reserved for it
  - utlb miss handler can itself fault—allow paged page tables

- OS is free to choose page table format!
MIPS Memory Layout

FFFF FFFF
C000 0000
BFFF FFFF
A000 0000
9FFF FFFF
8000 0000
7FFF FFFF

kseg2: Paged Kernel
kseg1: Phys. Uncached
kseg0: Phys. Cached

Kernel Memory

useg: Paged User

User Memory
Paging in day-to-day use

• Paging Examples
  - Demand paging
  - Growing the stack
  - BSS page allocation
  - Shared text
  - Shared libraries
  - Shared memory
  - Copy-on-write (fork, mmap, etc.)

• Next time: detailed discussion on MIPS